Continuations, Processes, and Sharing

Paul Downen, Luke Maurer, Zena M. Ariola, Daniele Varacca

University of Oregon, Université Paris Diderot

September 8, 2014

・ロト ・ 日 ・ ・ 日 ・ ・ 日 ・ ・ つ へ ()

The plethora of semantic artifacts

- Many ways to understand programming languages:
 - small-step semantics
 - big-step semantics (natural semantics)
 - abstract machines
 - continuation-passing style transformations

- •
- Different tools; different views
 - High-level reasoning
 - Low-level reasoning
 - Proof development

Q But which one to choose?

▲□▶ ▲圖▶ ▲ 臣▶ ▲ 臣▶ ― 臣 … のへぐ

Q But which one to choose?

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●

A All of them!

Q But which one to choose?

A All of them!

Q But how do we know that they agree?

- **Q** But which one to choose?
- A All of them!
- **Q** But how do we know that they agree?
- A Systematic inter-derivation; correct by construction (Danvy et al.)

ション ふゆ く 山 マ チャット しょうくしゃ

Processes as a semantic tool

- Embedding into processes (π -calculus)
- Computation as communication
- Strong resemblance to continuation-passing

Processes as a semantic tool

- Embedding into processes (π -calculus)
- Computation as communication
- Strong resemblance to continuation-passing

What do processes have to offer?

- \blacktriangleright Some computations more direct in $\pi\text{-}$ than $\lambda\text{-}calculus$
 - Concurrency
 - Non-determinism
 - Change over time
- Simple story for memoization
- Reveals techniques in implementations of lazy languages

ション ふゆ く 山 マ チャット しょうくしゃ

"Black holes" in GHC

From continuations to processes

Example: function composition

Transform the function composition

f (g 1)

into a process



Name intermediate results of serious computations:

f (g 1)

goes to

let $y = g \ 1$ in $f \ y$

◆□▶ ◆□▶ ◆臣▶ ◆臣▶ 三臣 - のへ⊙

Continuation-passing style

Rewrite into continuations:

$$\mathbf{let} \ y = g \ 1 \ \mathbf{in} \ f \ y$$

goes to

$$g(1, (\lambda y. f(y, \mathbf{ret})))$$

▲□▶ ▲圖▶ ▲臣▶ ★臣▶ ―臣 …の�?

Name all serious values:

$$g(1, (\lambda y. f(y, \mathbf{ret})))$$

goes to

let
$$k = \lambda y$$
. $f(y, ret)$ in $g(1, k)$

Rewrite into explicit environment:

let
$$k = \lambda y. f(y, ret)$$
 in $g(1, k)$

goes to

$$\nu k. k \coloneqq \lambda y. f(y, \mathbf{ret}) \text{ in } g(1, k)$$

▲□▶ ▲圖▶ ▲臣▶ ★臣▶ ―臣 …の�?

Rewrite into processes:

$$\nu k. k \coloneqq \lambda y. f(y, \mathbf{ret}) \text{ in } g(1, k)$$

goes to

$$\boldsymbol{\nu} k \left(! k(\boldsymbol{y}) . \, \overline{f} \langle \boldsymbol{y}, \mathbf{ret} \rangle \, | \, \overline{g} \langle 1, k \rangle \right)$$

◆□▶ ◆圖▶ ◆臣▶ ◆臣▶ 臣 - のへで

▲□▶ ▲圖▶ ▲目▶ ▲目▶ 目 のへの

$\begin{aligned} & \mathsf{Call-by-value} \\ & \mathcal{C}\llbracket MN \rrbracket \triangleq \lambda k. \, \mathcal{C}\llbracket M \rrbracket (\lambda v. \, \mathcal{C}\llbracket N \rrbracket \ (\lambda w. \, v \ (\lambda k'. \, k'w, \, k))) \end{aligned}$

$\begin{aligned} & \mathsf{Call-by-name} \\ & \mathcal{C}\llbracket MN \rrbracket \triangleq \lambda k. \, \mathcal{C}\llbracket M \rrbracket (\lambda v. \, v \, (\lambda k'. \, \mathcal{C}\llbracket N \rrbracket k', k)) \end{aligned}$

 $\mathcal{C}[\![x]\!] \triangleq \lambda k. \times k$ $\mathcal{C}[\![\lambda x. M]\!] \triangleq \lambda k. k \ (\lambda(x, k'). \mathcal{C}[\![M]\!] k')$

Uniform CPS transform (CBN and CBV)

Uniform CPS to Uniform π -encoding

$$\mathcal{C}[\![x]\!]k = x \ k$$
$$\mathcal{N} \circ \mathcal{C}[\![x]\!]k = x \ k$$
$$\mathcal{P} \circ \mathcal{N} \circ \mathcal{C}[\![x]\!]k = \overline{x} \langle k \rangle$$

$$\mathcal{C}\llbracket\lambda x. M\rrbracket k = k \ (\lambda(x, k'). \mathcal{C}\llbracketM\rrbracket k')$$
$$\mathcal{N} \circ \mathcal{C}\llbracket\lambda x. M\rrbracket k = \nu f. f := \lambda(x, k'). \mathcal{N} \circ \mathcal{C}\llbracketM\rrbracket k' \text{ in } k f$$
$$\mathcal{P} \circ \mathcal{N} \circ \mathcal{C}\llbracket\lambda x. M\rrbracket k = \nu f \ (!f(x, k'). \mathcal{P} \circ \mathcal{N} \circ \mathcal{C}\llbracketM\rrbracket k' \mid \overline{k}\langle f \rangle)$$

. . .

▲□▶ ▲圖▶ ▲ 臣▶ ▲ 臣▶ ― 臣 … のへぐ

Interlude: Of variables and values

▲□▶ ▲圖▶ ▲臣▶ ★臣▶ ―臣 …の�?

A mismatch

Soundness: steps in source are steps in target

$$(\lambda x. \lambda y. y)z = \lambda y. y$$

This is invalid by CBV transform of application:

$$\mathcal{C}\llbracket(\lambda x. \lambda y. y)z\rrbracket = \lambda k. z \ (\lambda w. \mathcal{C}\llbracket\lambda y. y\rrbracket k)$$

$$\neq \mathcal{C}\llbracket\lambda y. y\rrbracket$$

Variables are not values

$$\mathcal{C}[\![x]\!] \triangleq \lambda k. \mathbf{x} \ k$$

In CBN we need to execute a computation In CBV we need to lookup or fetch the value

Plotkin CBVRestricted CBVValues: $V ::= x \mid \lambda x. M$ $V ::= \lambda x. M$ Evaluation Contexts: $E ::= [] \mid EM \mid VE$ $E ::= [] \mid EM \mid VE$

 $(\lambda x. M)V = M\{V/x\}$ (β_v)

ション ふゆ く 山 マ チャット しょうくしゃ

Correctness bisimulation

▲□▶ ▲圖▶ ▲ 臣▶ ▲ 臣▶ ― 臣 … のへぐ

A transformation should preserve observable results of a program:

► Termination: the program reaches an answer

- Divergence: the program loops forever
- Getting stuck: the program cannot proceed

ション ふゆ く 山 マ チャット しょうくしゃ

How the proof should go

▶ ...

T[[·]] preserves immediate results
 If *M* is an answer then *T*[[*M*]] is an answer

• $\mathcal{T}[\![\cdot]\!]$ preserves reduction

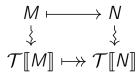
$$\mathcal{T}[\![M]\!] \longmapsto \mathcal{T}[\![N]\!]$$

How the proof actually goes

▶ ...

T[[·]] preserves immediate results
 If *M* is an answer then *T*[[*M*]] reaches an answer

• $\mathcal{T}[\![\cdot]\!]$ preserves reduction

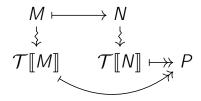


How the proof actually goes

▶ ...

T[[·]] preserves immediate results
 If *M* is an answer then *T*[*M*] reaches an answer

• $\mathcal{T}[\cdot]$ preserves reduction?

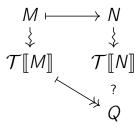


How the proof actually goes

▶ . . .

- $\mathcal{T}[\![\cdot]\!]$ preserves immediate results
 - ▶ If *M* is an answer then $\mathcal{T}[\![M]\!]$ reaches an answer

• $\mathcal{T}[\![\cdot]\!]$ preserves reduction???



Out-of-synch computations: administration

Source:

$$(\lambda x. x)(\lambda y. y) \longmapsto \lambda y. y$$

Target:

$$\mathcal{C}\llbracket (\lambda x. x)(\lambda y. y) \rrbracket \mathbf{ret} \longmapsto \mathbf{ret} (\lambda(y, k). (\lambda k'. y \ k') \ k)$$

 $\mathcal{C}\llbracket\lambda y. y
rbracket$ ret $(\lambda(y, k). (\lambda k'. y k') k)$

・ロト ・ 日 ・ ・ 日 ・ ・ 日 ・ ・ つ へ ()

Out-of-synch computations: aliasing

Source:

$$(\lambda f.g(f,f))(\lambda x.x) \longmapsto g((\lambda x.x),(\lambda x.x))$$

Target:

$$\mathcal{N}\llbracket (\lambda f. g(f, f))(\lambda x. x) \rrbracket$$

= $\nu i. i \coloneqq \lambda x. x \text{ in } (\lambda f. g(f, f)) i$
 $\mapsto \nu i. i \coloneqq \lambda x. x \text{ in } g(i, i)$

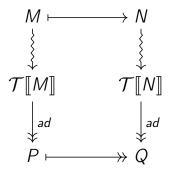
$$\mathcal{N}\llbracket g((\lambda x. x), (\lambda x. x)) \rrbracket$$

= $\nu i. i \coloneqq \lambda x. x$ in $\nu j. j \coloneqq \lambda x. x$ in $g(i, j)$

▲□▶ ▲圖▶ ▲臣▶ ★臣▶ ―臣 …の�?

Reasoning up to out-of-synch administration

Define an administrative free transform (Danvy and Nielsen TCS 2003)



◆□▶ ◆□▶ ◆□▶ ◆□▶ ● ● ●

Reasoning up to out-of-synch administration

Reason up to bisimulation

 $M \longmapsto N$

 \sim \sim



 $M \sim P$ iff $\mathcal{T}[M]$ ret $\longrightarrow_{ad} P$

・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・
 ・

Reasoning up to out-of-synch aliasing

Reason up to bisimulation







$$M\sim P$$
 iff $M\equiv \mathcal{N}^{-1}\langle\!\langle P
angle\!
angle$

▲□▶ ▲圖▶ ▲臣▶ ★臣▶ ―臣 …の�?

Bisimulation technique

Start out similar

M ∼ 𝒯[[M]]

Keep being similar

End up similar

 $\begin{array}{cccc} M \downarrow & & M \vdash \twoheadrightarrow N \downarrow \\ \sim & & \sim \\ P \vdash \twoheadrightarrow Q \downarrow & & Q \downarrow \end{array}$

One direction suffices

The forward direction sufficient if (Leroy):

- The source language is deterministic;
- ► No infinite loop in source terminates in target

Dichotomy of source reductions (Danvy and Zerny, PPDP'13) guarantees point 2:

- Proper reduction *must* cause work in target
- Administrative reduction *must* terminate

Sharing

◆□▶ ◆□▶ ◆臣▶ ◆臣▶ 臣 の�?

Call-by-need evaluation

let x = 1 + 2 in x * x

◆□▶ ◆□▶ ◆臣▶ ◆臣▶ 臣 の�?

let x = 1 + 2 in x * x

◆□▶ ◆□▶ ◆臣▶ ◆臣▶ 臣 の�?

let x = 1 + 2 in x * x

let x = 1 + 2 in x * x $\mapsto let x = 3 in x * x$

▲□▶ ▲圖▶ ▲臣▶ ★臣▶ ―臣 …の�?

$$let x = 1 + 2 in x * x$$

$$\mapsto let x = 3 in x * x$$

$$\mapsto let x = 3 in 3 * x$$

◆□▶ ◆□▶ ◆臣▶ ◆臣▶ 臣 のへで

$$let x = 1 + 2 in x * x$$

$$\mapsto let x = 3 in x * x$$

$$\mapsto let x = 3 in 3 * x$$

▲□▶ ▲圖▶ ▲ 臣▶ ★ 臣▶ 三臣 … 釣�?

$$let x = 1 + 2 in x * x$$

$$\mapsto let x = 3 in x * x$$

$$\mapsto let x = 3 in 3 * x$$

<□▶ <□▶ < □▶ < □▶ < □▶ < □ > ○ < ○

$$let x = 1 + 2 in x * x$$

$$\mapsto let x = 3 in x * x$$

$$\mapsto let x = 3 in 3 * x$$

$$\mapsto let x = 3 in 3 * 3$$

◆□▶ ◆□▶ ◆臣▶ ◆臣▶ 臣 のへで

$$let x = 1 + 2 in x * x$$

$$\mapsto let x = 3 in x * x$$

$$\mapsto let x = 3 in 3 * x$$

$$\mapsto let x = 3 in 3 * 3$$

▲□▶ ▲圖▶ ▲ 臣▶ ▲ 臣▶ ― 臣 … のへぐ

$$let x = 1 + 2 in x * x$$

$$\mapsto let x = 3 in x * x$$

$$\mapsto let x = 3 in 3 * x$$

$$\mapsto let x = 3 in 3 * 3$$

$$\mapsto let x = 3 in 9$$

<□> <0</p>
<□> <0</p>
<0</p>
<0</p>

Call-by-need and stateful CPS

Okasaki call-by-need CPS using assignment

$$\mathcal{C}[\![x]\!] \triangleq \lambda k. x \ k$$
$$\mathcal{C}[\![\lambda x. M]\!] \triangleq \lambda k. k \ (\lambda(x, k'). \mathcal{C}[\![M]\!]k')$$
$$\mathcal{C}[\![MN]\!] \triangleq \lambda k. \mathcal{C}[\![M]\!](\lambda v. \nu x.$$
$$x := \operatorname{memo}_{x}(N) \operatorname{in} v(x, k))$$

 $\operatorname{memo}_{x}(N) \triangleq \lambda k. C[[N]](\lambda w. x \coloneqq (\lambda k'. k' w) \text{ in } k w)$

On liveness of variables

• First evaluate M, with N assigned to x

$$let x = N in M$$

When x is forced, evaluate N and x is no longer in scope

$$\det x = N$$
 in $E[x]$

When N becomes V, continue in body with V assigned to x

$$\operatorname{let} x = V \operatorname{in} E[V]$$

Constructive update

Initial binding is ephemeral, disappears on lookup

$$\nu f. f :=_1 \operatorname{memo}_f(N) \operatorname{in} f k$$
 $\longmapsto \nu f. \operatorname{memo}_f(N)k$

Updated binding is permanent, always available and can never be changed

$$\nu f. f := V \text{ in } f k$$
$$\longmapsto \nu f. f := V \text{ in } V k$$

(ロ) (型) (E) (E) (E) (O)

Call-by-need and constructive update CPS

Thunking protocol strictly enforced (thunks only evaluated once):

$$C[[x]] \triangleq \lambda k. x \ k$$

$$C[[\lambda x. M]] \triangleq \lambda k. k \ (\lambda(x, k'). C[[M]]k')$$

$$C[[MN]] \triangleq \lambda k. C[[M]](\lambda v. \nu x.$$

$$x :=_1 \operatorname{memo}_x(N) \operatorname{in} v(x, k))$$

 $\operatorname{memo}_{x}(N) \triangleq \lambda k. C[[N]](\lambda w. x \coloneqq (\lambda k'. k' w) \text{ in } k w)$

Constructive update in the π -calculus

Permanent assignment: replicated server

$$\mathcal{P}\llbracket x \coloneqq \lambda y. M \text{ in } N \rrbracket = !x(y). \mathcal{P}\llbracket M \rrbracket | \mathcal{P}\llbracket N \rrbracket$$

Ephemeral assignment: unreplicated server

$$\mathcal{P}\llbracket x \coloneqq_1 \lambda y. \ M \text{ in } N \rrbracket = x(y). \ \mathcal{P}\llbracket M \rrbracket \mid \mathcal{P}\llbracket N \rrbracket$$

Processes can now responsively change their behavior

(ロ) (型) (E) (E) (E) (O)

Constructive update in machines

- Suspended computations removed on retrieval (Sestoft)
- Thunks become dead when forced
 "Black holes" in GHC
- Practical implementation techniques reflected by theory

ション ふゆ アメリア メリア しょうくしゃ

Conclusions

Program transformation from CPS to processes

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●

- Unite CBN and CBV with CBNeed
- Constructive update model of memoization
- CPS λ -calculus for change over time

Conclusions

- Program transformation from CPS to processes
- Unite CBN and CBV with CBNeed
- Constructive update model of memoization
- CPS λ -calculus for change over time
- Reminder: Decisions have consequences; live with them

Questions?

<□▶ <□▶ < □▶ < □▶ < □▶ < □ > ○ < ○